Abstract Presentation Back to Agenda Next Abstract

Building the Support for Radar Processing across Memory Hierarchies: On the Development of an Array Class with Shapes using Expression Templates in C^{++} [*]

Lenore R. Mullin Xingmin Luo[†] Lawrence A. Bush[‡]

August 7, 2003

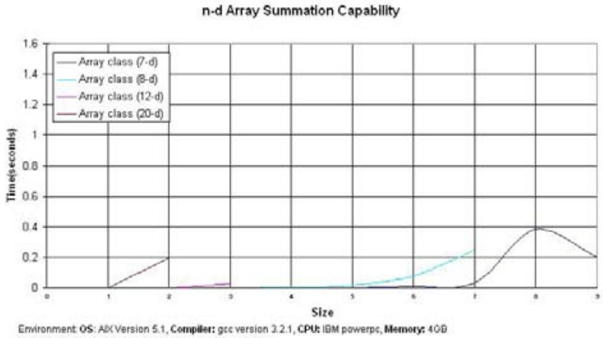
Abstract

"Embedded software processing requirements for DSP, especially for radar, are expected to exceed 1×10^{12} operations per second within five years [3]." Therefore, the efficient use of memory at all levels of the hierarchy is essential. These array based computations involve the composition of linear and multi-linear operators. Previous work illustrated how a general array algebra (MoA), and a "suitably rich compatible index calculus [3]" (Psi-Calculus), could be used to develop software for radar and other DSP applications. This software needs to be tuned to use the levels of memory hierarchies efficiently without the materialization of array valued temporaries [3]. Monolithic compiler experiments presented in [4] illustrated how these theories could be mechanized using expression templates in C^{++}. The present work continues these investigations by defining an N-dimensional `array class` with shape in order to support the mechanization of linear transformations in the Psi-Calculus (ψ-Calculus). We show that this class extends the support for array operations in the Portable Expression Template Engine ($<$ **PETE**$>$) while offering performance that is competitive with hand coded C. Such extensions are needed to support the dimension lifting which maps arrays to all levels of a memory/processor hierarchy.

Keywords: embedded digital systems, radar, signal processing, arrays, high performance, index calculus, shapes, psi, MoA.

Introduction

Motivating this paper is the development of efficient algorithms for radar and more generally DSP applications. "Reasoning about radar, from a *computational perspective*, entails reasoning about the data structures underlying the algorithms for radar computations [3]." Arrays are the data structures underlying algorithms for radar computations. These algorithms are characterized by linear and multi-linear matrix operations. Therefore, a high level array algebra can facilitate an efficient, scalable and portable algorithm design. "Consequently, we believe that the future development of efficient, scalable, portable algorithms, for radar, more generally for DSP applications, will be greatly facilitated by the use of a high-level array algebra during algorithm design. Additionally, since program efficiency depends critically upon the efficient use of memory/processor hierarchies, this array algebra should be combined with a suitably powerful index calculus. This calculus should facilitate data layout, movement, and manipulation at all levels of the memory/processor hierarchy. [3]." In [5, 1] it is shown that MoA and ψ-Calculus are suitable for such an algebra and calculus. For example, [1] presents in detail how MoA and the ψ-Calculus can be integrated into a Time Domain (TD) convolution algorithm. The algorithm development presented in [1] entailed array dimension lifting and data restructuring. These were driven by the memory/processor hierarchy, coincident with array decompositions and layouts. This process was shown to minimize temporary array materializations using these theories. Prior to this, compiler experiments were presented in [4] which demonstrated the machanization of these theories via expression templates in C^{++}.



Environment: **OS:** AIX Version 5.1, **Compiler:** gcc version 3.2.1, **CPU:** IBM powerpc, **Memory:** 4GB
Note: Array class supports any data type and dimemsion.

Figure 1: `Array class` with shape for N-dimensional support.

TD Convolution

Synthetic aperture radar (SAR) has many application (i.e. target detection, continuous observation of dynamic phenomena and classification of vegetation [6]). This is due to its high resolution imaging capabilities under varying conditions (day and night, all-weather). Various SAR signal processing methods have been developed (i.e. spectral analysis (FFT) and frequency domain convolution). However, the time-domain (TD) analysis is the simplest and most accurate algorithm for SAR signal processing [6]. The TD algorithm is also the most computational intensive which makes it useful only with SAR data of limited size [6]. Consequently, faster TD algorithms are needed as the size and resolution requirements increase.

TD Convolution: MoA Design

In [4] a C^{++} vector class to define the TD convolution was presented. The related experiments showed that the creation of array valued temporaries could be avoided, enabling performance competitive with hand coded C.

A uni-processor using vector arguments on the vector class presented in [4] was sufficient for those experiments. However, to support mapping to processor memory hierarchies, vector arguments must be algebraically abstracted to higher dimensional arrays. When processors are added to the design, the dimension of the problem is lifted up. Adding a cache loop adds yet another dimension. Thus, we started with a 1-dimensional problem, then abstracted the computation to a second (time) dimension. Adding processor and cache mapping ultimately resulted in a 4-dimensional problem. In addition, if we desire to support 3-dimensional or higher array arguments, dimension lifting may require 10 or more dimensions. Such a high dimensionality is not typically supported in today's languages or libraries. Figure 1 illustrates our ability to do so.

Shape : a new `class`

$<$ **PETE**$>$ facilitates the creation of optimized C^{++} code to do various mathematical operations. However, $<$ **PETE**$>$ operates on scalars and does not provide an interface for multi-dimensional array computations that are required for the ψ-Calculus. In addition to the dimension limitation, $<$ **PETE**$>$ does not support

[*] NSF CCR 0105536

[†] Department of Computer Science, University of Albany, State University of New York, Albany, NY 12222, U.S.A.`{lenore, xluo}@cs.albany.edu`

[‡] Department of Computer Science, Rensselaer Polytechnic Institute, Troy, NY 12180, U.S.A. `BushL2@cs.rpi.edu`

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE **20 AUG 2004**	2. REPORT TYPE **N/A**	3. DATES COVERED **-**

4. TITLE AND SUBTITLE
Building the Support for Radar Processing across Memory Hierarchies: On the Development of an Array Class with Shapes using Expression Templates in C++

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Department of Computer Science, University of Albany, State University of New York, Albany, NY 12222, U.S.A. 12222; Department of Computer Science, Rensselaer Polytechnic Institute, Troy, NY 12180, U.S.A.

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release, distribution unlimited

13. SUPPLEMENTARY NOTES
See also ADM001694, HPEC-6-Vol 1 ESC-TR-2003-081; High Performance Embedded Computing (HPEC) Workshop (7th)., The original document contains color images.

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**UU**	**17**	

Abstract

Presentation

Back to Agenda

Next Abstract

Paper: **Building the Support for Radar Processing Across Memory Hierarchies:**

On the Development of an Array Class with Shape using C++ Expression Templates

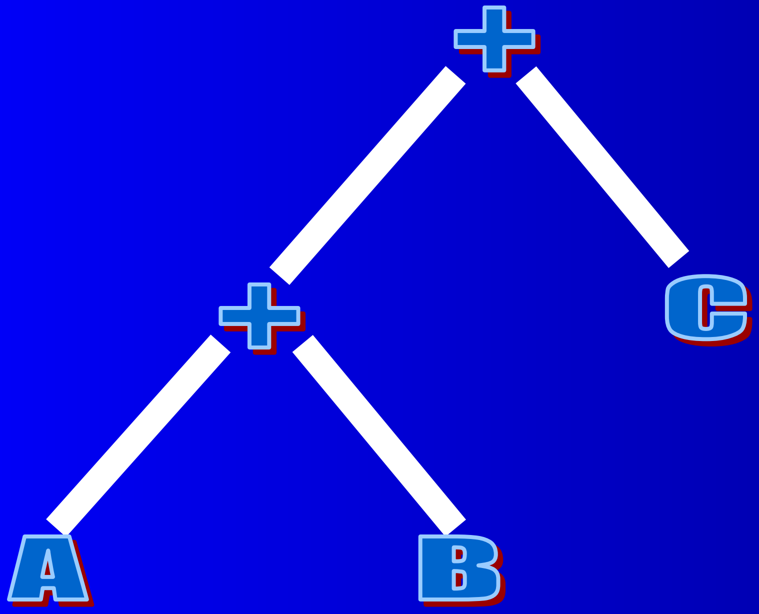
by Lenore R. Mullin, Xingmin Luo
and Lawrence A. Bush

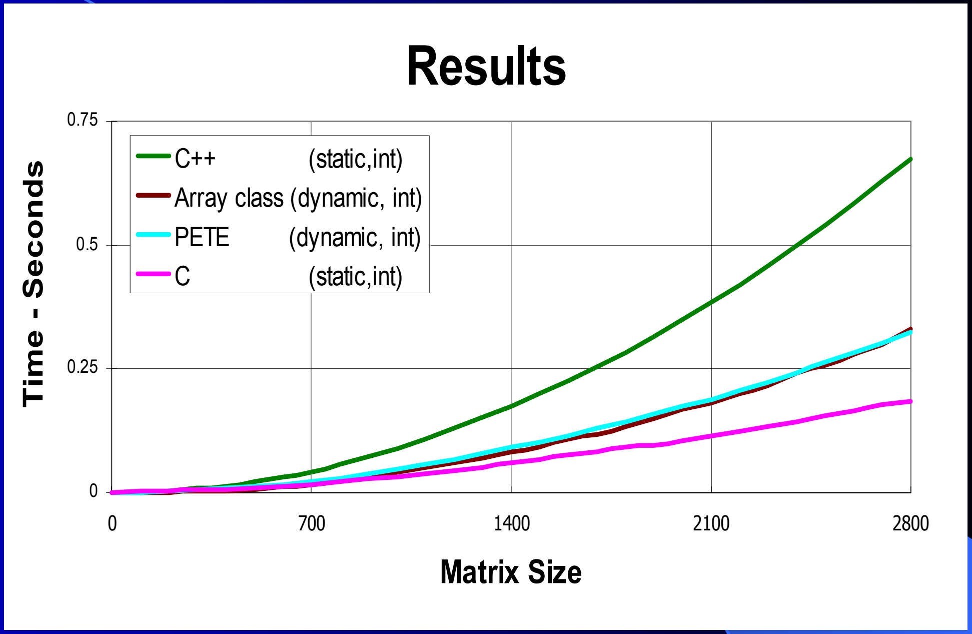
Processor / Memory Mapping

$$\begin{array}{c} \sum_{l=0}^{\frac{(T\,\vec{H})-1}{CACHE}} \sum_{l=0}^{(T\,\vec{H})-1} \\ \sum_{l=0}^{\frac{(T\,\vec{H})-1}{CACHE}} \sum_{l=0}^{(T\,\vec{H})-1} \\ \vdots \\ \sum_{l=0}^{\frac{(T\,\vec{H})-1}{CACHE}} \sum_{l=0}^{(T\,\vec{H})-1} \end{array} \begin{pmatrix} \begin{pmatrix} <15\ 14\ 13> \\ <12\ 11\ 10> \end{pmatrix} \\ \begin{pmatrix} <15\ 14\ 13> \\ <12\ 11\ 10> \end{pmatrix} \\ \vdots \\ \begin{pmatrix} <15\ 14\ 13> \\ <12\ 11\ 10> \end{pmatrix} \end{pmatrix} \times \begin{pmatrix} \begin{pmatrix} <0\ 0\ 0> \\ <0\ 0\ 1> \end{pmatrix} \\ \begin{pmatrix} <0\ 0\ 0> \\ <0\ 1\ 2> \end{pmatrix} \\ \vdots \\ \begin{pmatrix} <1\ 2\ 3> \\ <4\ 5\ 6> \end{pmatrix} \end{pmatrix} \longmapsto \text{processor } 0$$

$$\begin{array}{c} \sum_{l=0}^{\frac{(T\,\vec{H})-1}{CACHE}} \sum_{l=0}^{(T\,\vec{H})-1} \\ \sum_{l=0}^{\frac{(T\,\vec{H})-1}{CACHE}} \sum_{l=0}^{(T\,\vec{H})-1} \\ \vdots \\ \sum_{l=0}^{\frac{(T\,\vec{H})-1}{CACHE}} \sum_{l=0}^{(T\,\vec{H})-1} \end{array} \begin{pmatrix} \begin{pmatrix} <15\ 14\ 13> \\ <12\ 11\ 10> \end{pmatrix} \\ \begin{pmatrix} <15\ 14\ 13> \\ <12\ 11\ 10> \end{pmatrix} \\ \vdots \\ \begin{pmatrix} <15\ 14\ 13> \\ <12\ 11\ 10> \end{pmatrix} \end{pmatrix} \times \begin{pmatrix} \begin{pmatrix} <2\ 3\ 4> \\ <5\ 6\ 7> \end{pmatrix} \\ \begin{pmatrix} <3\ 4\ 5> \\ <6\ 7\ 8> \end{pmatrix} \\ \vdots \\ \begin{pmatrix} <9\ 0\ 0> \\ <0\ 0\ 0> \end{pmatrix} \end{pmatrix} \longmapsto \text{processor } 1$$

<PETE> Shape

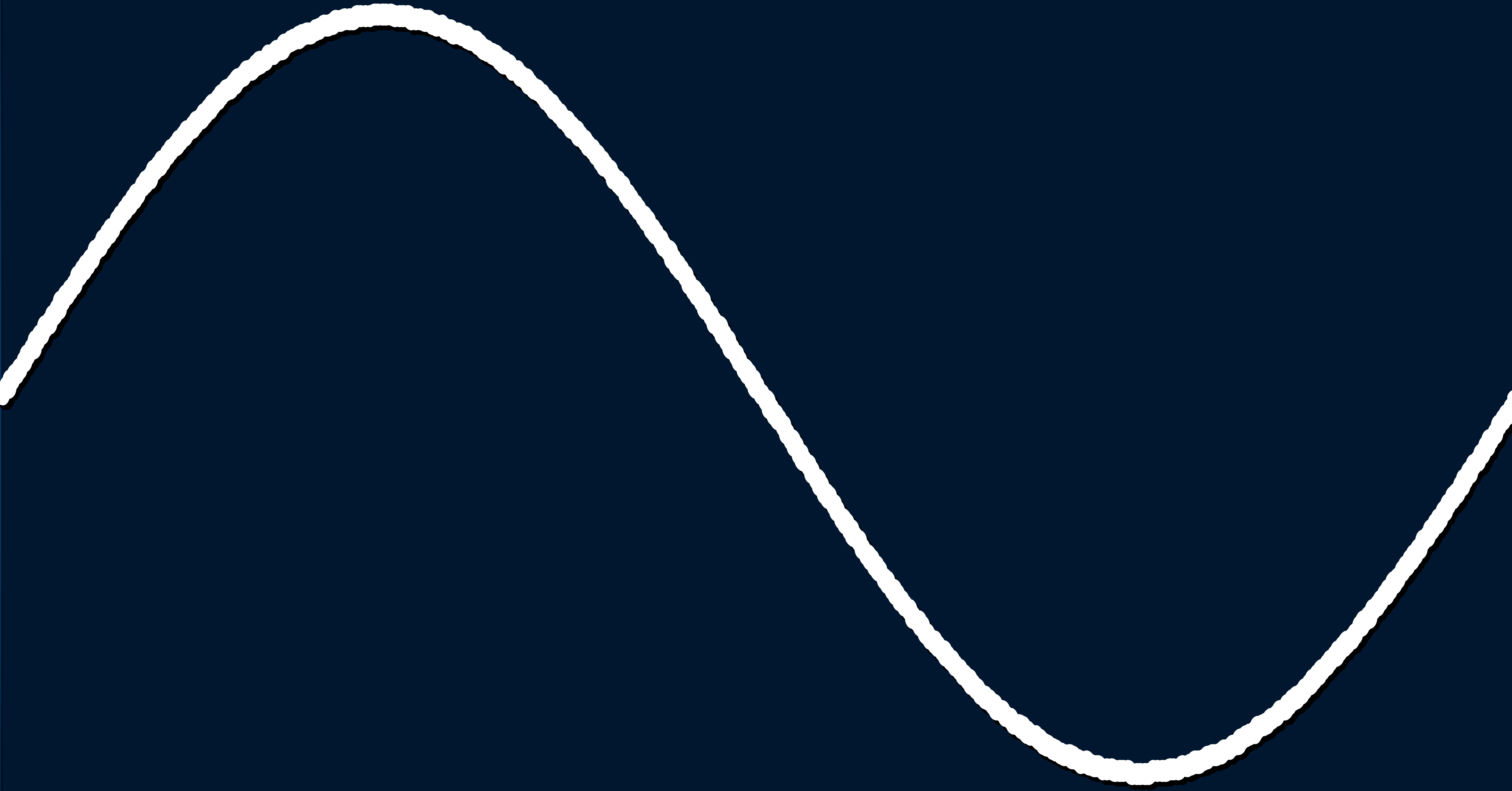




```
const Expression    <BinaryNode<OpAdd,  Reference<Array> ,
                     BinaryNode<OpAdd, Reference<Array>  ,
                     Reference<Array> > > >      &expr = A + B + C
```

Programmability

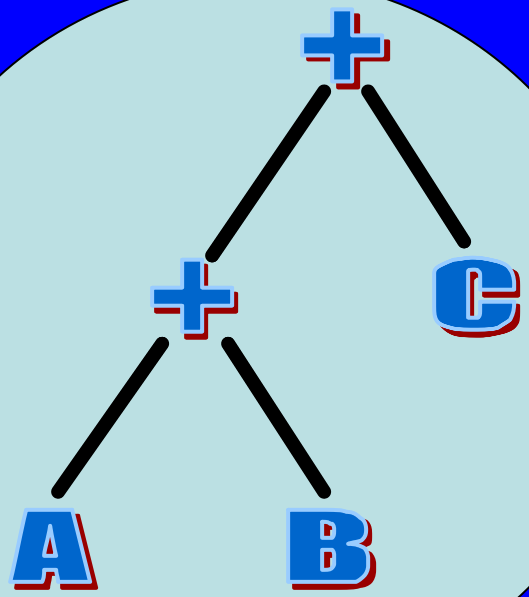
C++



Mechanization

<PETE>

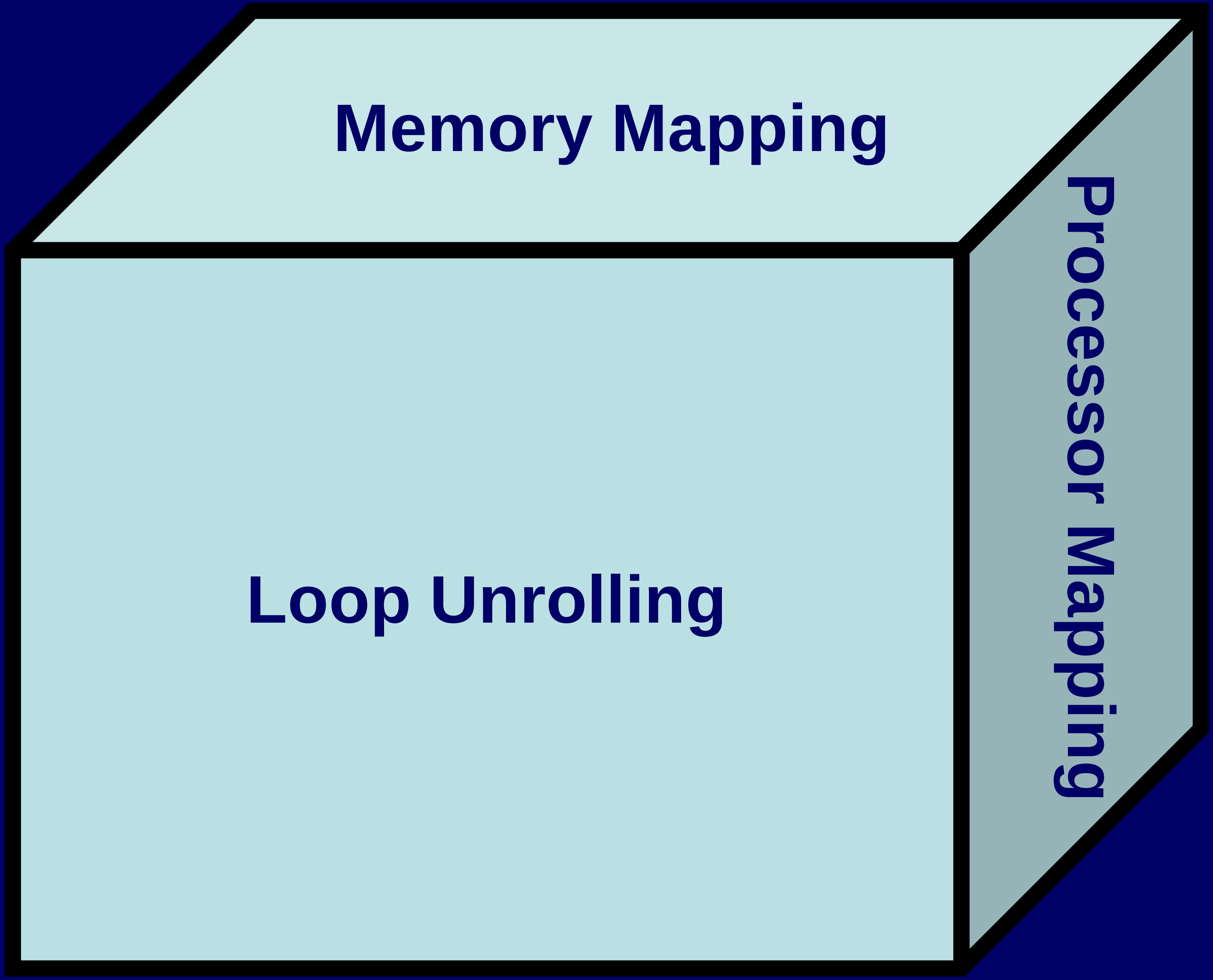
Portable
Expression
Template
Engine



Reduction Semantics

Psi - Calculus

Generalized Array Indexing

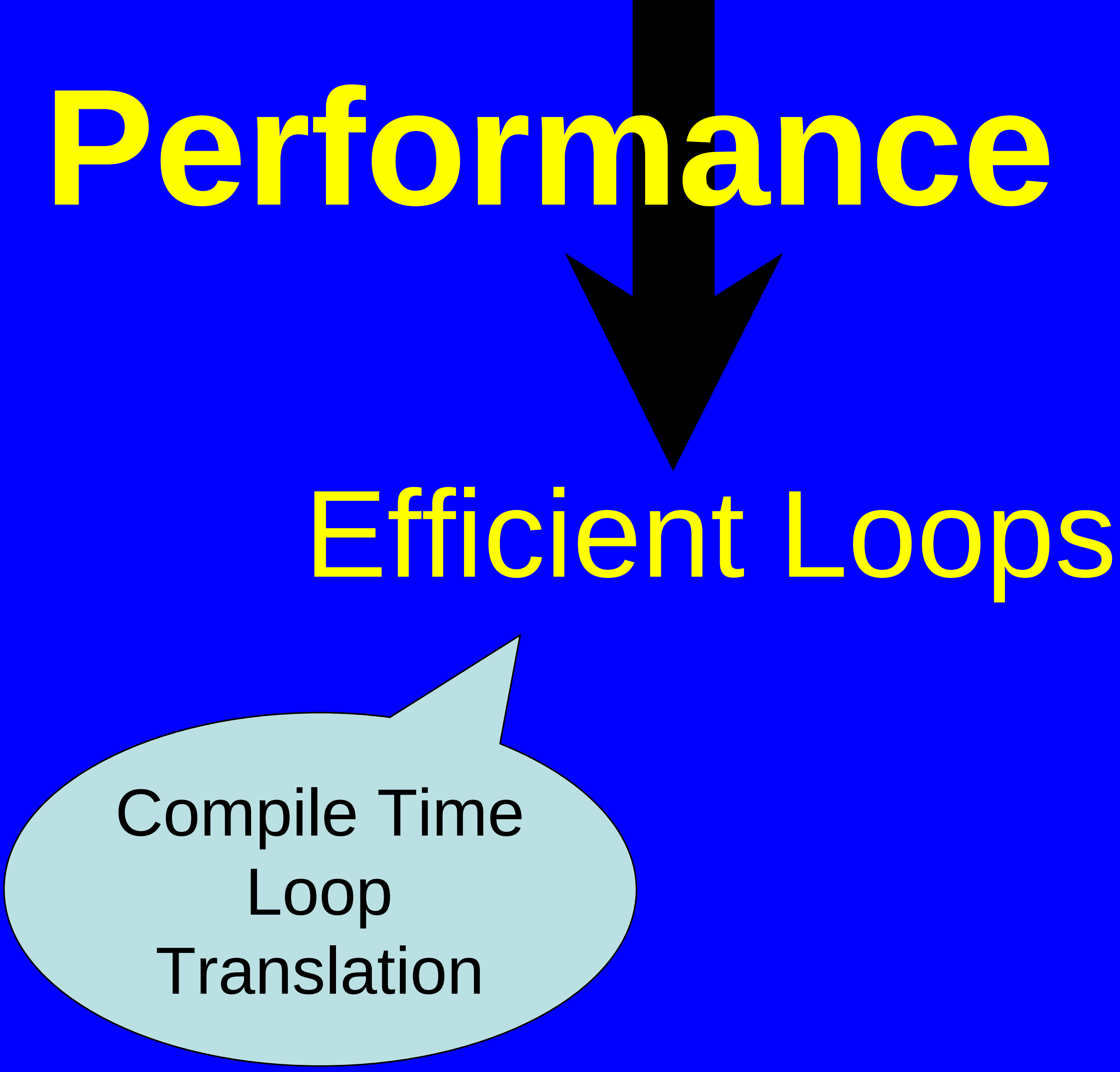


Memory Mapping

Loop Unrolling

Processor Mapping

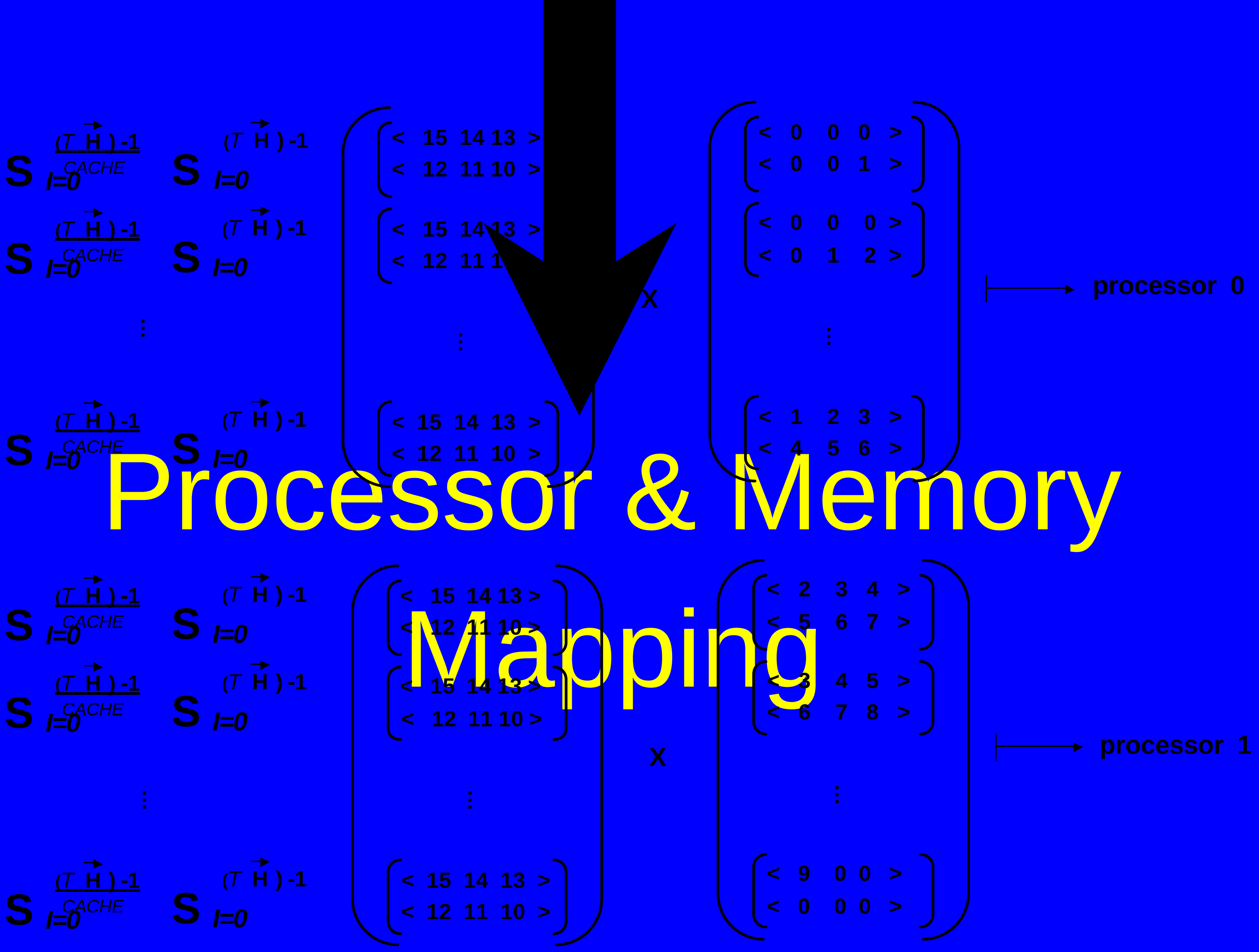
Performance



Efficient Loops

Compile Time
Loop
Translation

Processor & Memory Mapping



Operations

shift
take
rotate

Array.h

```
template <class T = int>
class Array
{
     . . .
     template<class RHS>
     Array &operator=(const Expression<RHS> &rhs)
     {
          for(long i=0; i<this->size; i++)
              d[i] = forEach(rhs, EvalLeaf1(i), OpCombine());
          return *this;     //equivalent to: a.d[i] = b.d[i]+c.d[i]+d.d[i]
     }
     . . .
     private:
          T * d;
          vector <int> shape;
          long size;
}
```

Array.h

integrates
with
<PETE>

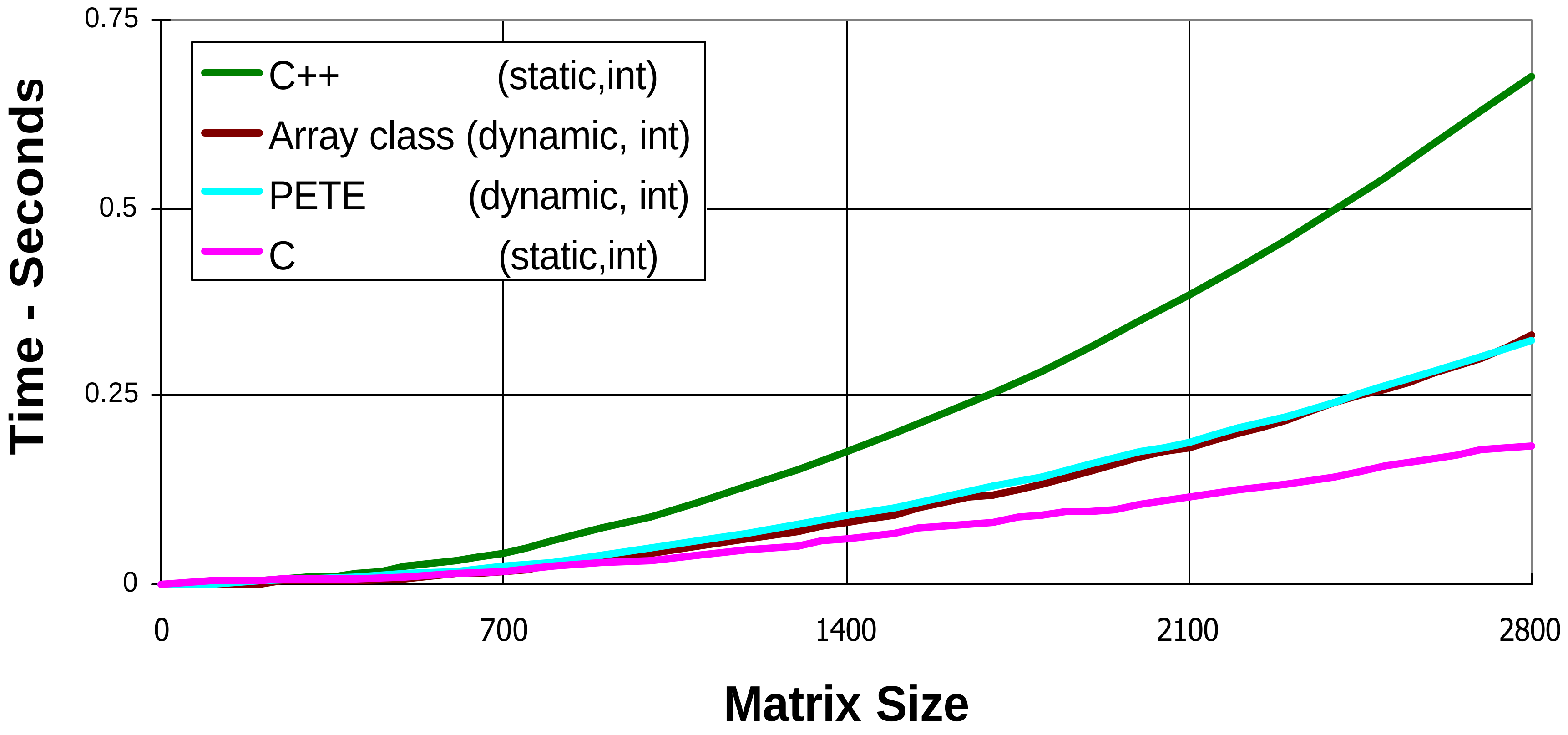
Psi-Calculus
platform

N – dimensional capability

(required for processor /
memory mapping)

Array.h

Results : Comparable to Hand Coded C



Abstract Presentation Back to Agenda Next Abstract

the shape notion which is a component used to calculate the attribute evaluation rules needed to rewrite an Abstract Syntax Tree (AST) defining array expressions. Fortunately, <**PETE**> is designed to be extendable. To that end, we have implemented a multi-dimensional array object extension to support shape. The shape vector uses the Standard Template Library (STL) `vector<int>` class which conveniently enables the needed N-dimensionality. The shape vector is passed to a specialized `Array` class which constructs the multi-dimensional array and enables assignments and arithmetic using operator overloading and expression templates respectively.

Our experiments test the efficiency of our implementation and show that ours is much faster than a standard C^{++} implementation and is similar in speed to an implementation in hand-coded C. It also shows that the N-dimensional functionality of our `Array class` does not increase the overhead compared to a typical PETE implementation.

Implementation

In a previous paper [4], experimental results were presented for computing 1-Dimensional Arrays. To demonstrate our efforts to extend these ideas to multi-dimensional arrays, we present experimental results of our multi-dimensional `Array` object implementation. The class is templated and therefore supports any data type. This is necessary so that we can use common scientific computing data types such as `float` and `double`. The experiments, therefore, test the `int`eger and `float` data types for basic math operations which are fundamental to ψ-Calculus (i.e. distributing indexing over scalar operations).

The results presented here are for the addition[1] of three multi-dimensional arrays and the assignment of their result to a forth multi-dimensional array.

The `Array` is defined through a shape vector. This vector stores the size of each dimension of an `Array` and is passed to the `Array` class constructor. Default and copy constructors are also defined. This class incorporates operator overloading and <**PETE**> related expression tree definitions. Our class gets its efficiency by defining and implementing the high level operation of the expression template. This allows our class to interface with <**PETE**>, utilizing its standard mathematical operations and expression tree operation evaluation ordering. *The shape vector mentioned above is a relevant feature of* ψ*-Calculus.*

Essentially, the ψ-Calculus rules can be implemented at the iterator level. Basically, we have complex non-algebraic array expressions that can be reduced to memory access patterns. The multiple levels of indirection can be handled by the iterator abstraction (the pointer on steroids).

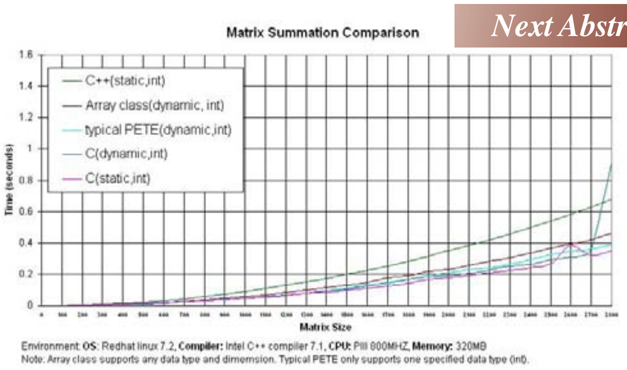
Experiments

Our tests were compiled on two COTS platforms: an 800 MHz Pentium III processor with 320MB of memory running Redhat Linux 7.2 and a 200 MHz IBM PowerPC with 4GB of memory running AIX Version 5. The test code was compiled using Intel C^{++} and GCC respectively.

Figure 2 and 3 show our results which measure the differences between six implementations of multi-dimensional array addition[2]: C^{++} (`static`, `int`), `Array` class (`dynamic`, `float`), `Array` class (`dynamic`, `int`), <**PETE**> (`dynamic`, `int`), C (`dynamic`, `int`) and C (`static`, `int`).

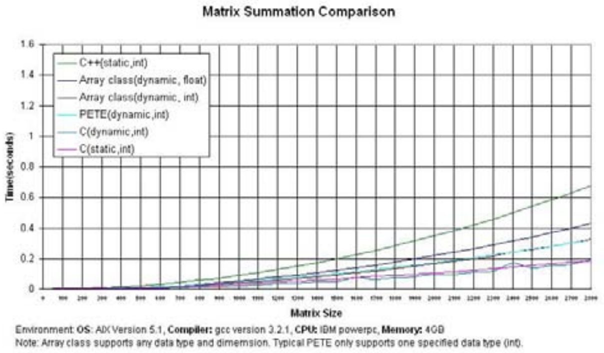
The results were similar on both platforms. Specifically, the C (`static`, `int`) version performed the fastest and the C^{++} (`static`, `int`) version performed the slowest. It is clear that the Object Oriented Programming (OOP) constructs of C^{++} affect performance. However, it is well known that OOP constructs improve programmability (ease of use, extendibility, reusability and quality [2]) over C, in particular when extending the implementation to complex applications. With <**PETE**> and our C^{++} `Array` class, we achieved similar performance as is obtained using C where the loops are optimized by hand. This is shown by the results using the `int`eger type which performed the same as the pure <**PETE**> coded results. However, the `float` data type version was still significantly faster than the traditional C^{++} (`static`, `int`) implementation. These results further validate that we can integrate the high performance of optimized C loops into the OO/C^{++} paradigm.

Overall, these experiments show that extending <**PETE**> to N-Dimensional Array operations via the ψ-Calculus shape notion is viable. The speed of the computations is impressive even when using templated types including the `float` data type.



Environment: **OS:** Redhat linux 7.2, **Compiler:** Intel C++ compiler 7.1, **CPU:** PIII 800MHZ, **Memory:** 320MB
Note: Array class supports any data type and dimension. Typical PETE only supports one specified data type (int).

Figure 2: Five implementations of multi-dimensional addition.



Environment: **OS:** AIX Version 5.1, **Compiler:** gcc version 3.2.1, **CPU:** IBM powerpc, **Memory:** 4GB
Note: Array class supports any data type and dimension. Typical PETE only supports one specified data type (int).

Figure 3: Six implementations of multi-dimensional addition.

Conclusion and Future Work

The results shown demonstrate the viability of using <**PETE**> as a means to optimize operations that are essential to a fully functional ψ-Calculus library. They represent a very important step: inclusion of the shape notion into the `Array class`.

These results are encouraging. Future work may be in reducing the detrimental performance caused by templating the `Array` type and adding additional algorithm methods to enable ψ-Calculus operations.

References

[1] H. B. Hunt III, L. Mullin, and D. J. Rosenkrantz. Towards an indexing calculus for efficient distributed array computation. Technical Report 97–4, University at Albany, Department of Computer Science, 1997.

[2] B. Meyer. *Object-Oriented Software Construction, Second Edition*. Prentice Hall PTR, Upper Saddle River, New Jersey, 1997.

[3] L. Mullin, H. H. III, D. Rosenkrantz, and X. Luo. Efficient radar processing via array and index algebras. In *Proceedings of the 1st Workshop on Optimizations for DSP and Embedded Systems(ODES)*, 2003.

[4] L. Mullin, E. Rutledge, and R. Bond. Monolithic compiler experiments using C++ Expression Templates. In *Proceedings of the High Performance Embedded Computing Workshop HPEC 2002*, MIT Lincoln Laboratory, Lexington, MA, 2002.

[5] L. M. R. Mullin. *A Mathematics of Arrays*. PhD thesis, Syracuse University, Dec. 1988.

[6] B. Wu, Y. Zhang, and J. Kong. Time-domain computer simulation of Synthetic Apeture Radar(SAR) image for rough surface. In *Proceedings of Progress in Electromagnetics Research Symposium*. Research Laboratory of Electronics, Massachussetts Institute of Technology, Cambridge, MA, July 2000.

[1] All <**PETE**> scalar operations are included.

[2] This could be any binary scalar operation: divide, ceiling, etc.